Telling Stories

1

Intent on a great project to renew the earth, God calls upon Noah, the one man he can trust to carry out his plans. He starts by clearly explaining the problem at hand:

> *And God said unto Noah, The end of all flesh is come before me; for the earth is filled with violence through them; and, behold, I will destroy them with the earth.*[1]

A forceful and concise communicator, God then details what he wants done in the first phase of the project:

> *Make thee an ark of gopher wood; rooms shalt thou make in the ark, and shalt pitch it within and without with pitch.*
>
> *And this is the fashion which thou shalt make it of: The length of the ark shall be three hundred cubits, the breadth of it fifty cubits, and the height of it thirty cubits.*
>
> *A window shalt thou make to the ark, and in a cubit shalt thou finish it above; and the door of the ark shalt thou set in the side thereof; with lower, second, and third stories shalt thou make it.*

Having explained the basic requirements of the ark, God moves on to the main processes of phase two of the project, beginning with his own action items:

> *And, behold, I, even I, do bring a flood of waters upon the earth, to destroy all flesh, wherein is the breath of life, from under heaven; and every thing that is in the earth shall die.*

COPYRIGHTED MATERIAL

[1] Genesis 6:13 (King James).

He then returns to what he requires of Noah:

> *But with thee will I establish my covenant; and thou shalt come into the ark, thou, and thy sons, and thy wife, and thy sons' wives with thee.*
>
> *And of every living thing of all flesh, two of every sort shalt thou bring into the ark, to keep them alive with thee; they shall be male and female.*
>
> *Of fowls after their kind, and of cattle after their kind, of every creeping thing of the earth after his kind, two of every sort shall come unto thee, to keep them alive.*

And mindful of the details, God makes sure Noah brings food sufficient to keep everyone fed through phase three:

> *And take thou unto thee of all food that is eaten, and thou shalt gather it to thee; and it shall be for food for thee, and for them.*

God doesn't just say, "Go and build an ark." He goes into a lot of detail about what he wants Noah to do. He also explains what is going to happen and why each thing is required, and he associates what is required of Noah with specific events in a logical order. We start with the general project goals, and then we learn about the ark; he warns of the flood, specifies the passenger list and boarding process, and finally, covers the food requirements.

Because God crafts a compelling narrative, Noah clearly understands what he is supposed to do and why, and the project is a success: " … all that God commanded him, so did he." We, too, still remember the details of Noah's requirements because they are part of a great story.

Of course God has a few advantages in the requirements process that the rest of us do not. We have to work a bit harder and go into more detail, and therein lies the main purpose of this book.

What Must *We* Do?

When you or I need someone to get something at the store, we write a shopping list. When we need a colleague to create a report on quarterly sales, we send an e-mail explaining the exact time period and figures we want. The shopping list and e-mail are simple *requirements documents*. You create a requirements document to explain what you need to someone capable of meeting the need.

If you're clear and precise, you'll probably get what you want. If there is any problem when the thing is done, you have a record of what you asked for and a credible basis for requesting corrections or improvements.

When you want something complicated, you must write a more detailed requirements document. And when you're explaining not only your own needs, but also the needs of a group, you must find a way to include the group of needers in your process and systematically find out what they want. (God didn't have this issue. He knew what he wanted in an ark and didn't have to reach consensus.) Once you've written something, you have to check back with the needers to make sure you are correctly expressing their needs. If the needers can't make sense of what you've written, they can't tell you if you got it right.

Just as God does with Noah, we do better if we explain what we want in plain language and a clear and logical sequence. We do better when we tell a story.

This book explains how to write a software requirements document using storytelling, the most ancient and human means for sharing information. I know you can't make a software requirements document into a story as exciting as Noah's Ark or *Huckleberry Finn*. But you can make a narrative that is engaging and easy to follow for the people who have an interest in solving the problem at hand. In the narrative, the problem should be clearly stated at the beginning, and all points should lead to the solution, with each point following from one to the next in a logical sequence of dependent outcomes. It also helps if you use a lot of pictures.

When you need something pretty ordinary that people have been making for a long time, like a chair or a hamburger, you don't usually have to explain what you want in much detail. There are perhaps a few variables you need to specify (with cheese, no fries, medium rare). It's more challenging when you need something complicated and specialized such as a new software system.

Only very skilled and specialized engineers make software, and they have gone to special schools to learn new languages and tools (Java, C#, and so on). These engineers mostly communicate with each other about what they make, using all the special words from these new languages, and pretty soon ordinary people can't understand them very well.

The challenge when trying to write a story about what you want software to do is to make it understandable to two groups that communicate in very different ways: the needers of the software and engineers who make it. If both groups do not understand and approve of your story, it will fail.

Why Do We Learn Better From Stories?

I wouldn't need to explain this to Aesop, the mythical author of *Aesop's Fables*. When he wanted to encourage savings, he didn't lecture on the dangers of poverty, he made up the story of the ant and the grasshopper. There are stories to everything, and our brains are built to listen to them.

Listening to a story makes us experience everything we hear as if we were part of the action. We relate to each event and fill in any details that might be left out. We are able to evaluate and remember each piece of information because it is part of a logical and realistic whole that we validate against our own experience. We compare the story to what we know and if it is believable and the information is important to us, we find it compelling in a way that a simple recitation of the same information could never be.

Admittedly, the story of making or changing most software systems is not that interesting, but if you do your best to relate what the system does to an understandable narrative, it will be much more compelling. In particular, if you emphasize the main problem the project aims to solve and carry it through every stage of your work, it can bring life to detailed requirements and make readers pay attention to information that might otherwise be tedious.

How Do Story Elements Relate to Requirements?

You may remember from your English classes (if you took any) that a story has setting, conflict, characters, plot, point of view, and theme. Let's go through these and explore how each is analogous to a part of the requirements process:

- **Conflict:** The basic problem that you are hoping to solve is the central conflict to resolve in the requirements process. We don't often think of a lack of technology as a conflict, but it is a conflict between an old way of working and a new and better way in the future (we hope).
- **Theme:** The theme is the central concept underlying the solution in your document. Viewed another way, a theme can be a very big requirement, such as the general need for an automated way of validating data in your system.
- **Setting:** A setting is the place and time of the story. In a requirements document, it is important to explain the broader context of the problem you are trying to solve. This could include some general information

about your technology environment, business, industry, economic conditions, or any other general background information that might be useful. You probably don't need a lot about the setting in a requirements document, but a paragraph or two in the executive summary can be very helpful.

- **Plot:** The series of processes that occur in the current and future system are the plot of your requirements document. Like the plot in a story, the events happen in a certain order and the outcome of each affects the later ones. They are what make you eagerly turn pages.
- **Characters:** There are many types of characters in a requirements document: some are people, some are machines or programs. Any entity capable of action can be a character in a requirements document.
- **Point of view:** It can be very useful to take different points of view as you describe different processes, or the same process. Your ultimate goal is to provide an omniscient view that faithfully describes everything that happens and what everyone needs. But along the way, it could make sense to explain an order process from *both* the customer and the order-taker point of view, for example.

If we learn best from stories, and if story elements are analogous to elements of a software requirements document, how do we put this idea into practice and make a better requirements document? First, I have to clarify what I mean by *software requirements document*.

What Are *Software* Requirements, and Who Are They For?

A software requirements document explains a business problem and the requirements of a software solution from a user *and* a business perspective. It is precise and detailed so that engineers can figure out what to build and how to evaluate what they produce. It uses common language that the business users and managers on the project can understand. You won't know if the requirements are right unless the needers tell you so, and they can't tell you if they can't understand the document.

Here are some qualities of effective requirements:

- A requirements document states *what* is needed from a system in precise, measurable, and testable terms.

- A requirements document details the way a system interacts with the elements around it.
- A requirements document uses plain language understandable to software users and engineers alike.

There are some very common wrong ideas and misconceptions about requirements documents:

- A requirements document does not explain *how* a system is built.
- A requirements document does not describe the technology used to make a system.
- A requirements document does not detail the schedule, plans, costs, or resources required to make the system.

Beyond its immediate utility to the needers of the software and the engineers making it, a requirements document has many secondary purposes:

- It sells a project to senior management and other stakeholders.
- It demonstrates that a thorough and well-considered process has taken place.
- It preserves a team's reasoning so that future team members can understand how choices were made.
- It provides the basis for functional specifications and acceptance tests.
- It can be a starting point for user documentation.

Are Software Requirements Different from Business Requirements?

Not in this book. I use the term *software requirements* very broadly to mean any requirements for making or changing software systems. Most of what I cover applies to any type of requirements, but some analysts attach a great deal of meaning to the term *software* in the phrase *software requirements*. Sometimes you will hear a great effort made to distinguish software requirements from business requirements, technical requirements, functional requirements, nonfunctional requirements, minimum daily requirements, and so on.

Having tangled with several of these terms in various environments and projects, I can confidently tell you there is no clear consensus about what they mean and how they should be used. When engaged to work on any type of requirements, you should not assume you know what to do just because

someone told you to go write "business" requirements, "software" requirements, or any other type.

I have often heard "business" requirements defined as requirements written by a nontechnical "business-side" group and then delivered to a "technical" group that analyzes them and then devises "software" requirements that are more technical and detailed. But I have also heard "business" requirements used to describe all types of nontechnical requirements.

I don't think dividing requirements into many categories is a very useful exercise. Using the approach I recommend, distinctions aren't really important. When gathering requirements, it can help to think of meaningful categories that remind people of what they might have overlooked. These categories are usually more specific than abstract terms such as "software" and "business." Think about performance requirements, data requirements, usability requirements, availability requirements, or any other category that is meaningful to your project. I'll make more specific suggestions when I discuss gathering content. I think it's best to focus on telling the main story of the system; every type of requirement will come up along the way.

Why Projects Collapse (Without Detailed Requirements)

To understand how a project can break down without effective written requirements, let's take a look at a fictional, but very typical story.

When starting a project to make a new order processing system, the makers and needers of the software have a few meetings. The makers listen to the basic ideas of what the needers want and very quickly get into talking among themselves about how they are going to build something to satisfy the needers. The needers can't follow the discussion, but try to participate by smiling and nodding.

After retreating to their offices, the makers present the needers with something in writing about what they are going to do and ask the needers to approve it. They call it a requirements document, but it says more about *how* the makers are going to do their work than the details of *what* they are making. There are considerable details about costs and resources and schedules, but nothing clear about things such as how to apply the discount rate for preferred customers and whether a customer profile can be altered from the transaction entry screen by users without customer service entitlements.

The needers, who don't want to seem stupid or mean in challenging the knowledge of the makers, haplessly approve the "requirements" document and go about their business. Who has time to read a document like this? The first 50 pages are lists of data fields and data types followed by a printout of a PowerPoint presentation with bullet points that say "State of the Art," "Secure Solution," and "Peak Performance." It's hard to know what these mean, but they sound impressive.

Besides, the needers don't feel like they should have to do a lot of work at this point, because after all, they are paying someone to meet *their* needs; let the makers figure it out. And what they need is so obvious. Anyone with a decent background in accounting knows the basics of transaction processing and can adapt it to this line of business. The manager of the makers seems so clever and smiles nicely as if she understands exactly what to do.

The makers disappear for three weeks and come back with a prototype of what they call the *interface* to show the needers. None of the needers dare to ask what an interface is. The makers show the needers several pictures of screens and the needers get some idea of how they will use the new system. Someone asks how they can apply a discount rate for preferred customers and the makers say that will be added later. No one takes notes. There are several more meetings showing different versions. The same person asks about the discount rate again and the makers show them where the discount field is on the customer-profile screen. The needers are reasonably happy with what they see (when they're paying attention), even if it is missing some details. The system goes into production. Soon, a customer calls up angry because he was not given his usual discount.

It turns out that no one who enters transactions has access to the customer-profile screen where it's possible to add the discount rate, and it would raise regulatory issues about segregation of duties if they did. Without substantially changing the architecture of the whole system, the only way to apply the discount is with a post-sale credit that takes a full day at the end of the month for a staff member to manually calculate with spreadsheets. Not an optimal outcome.

Because the needers and makers did not understand each other well enough and didn't spend enough time on requirements, the system doesn't meet some fundamental needs.

What I just described happened last year. This year, senior management wants to clean up the mess and try to make the system that was supposed to fix everything actually work, so they've drafted you, the hapless project manager,

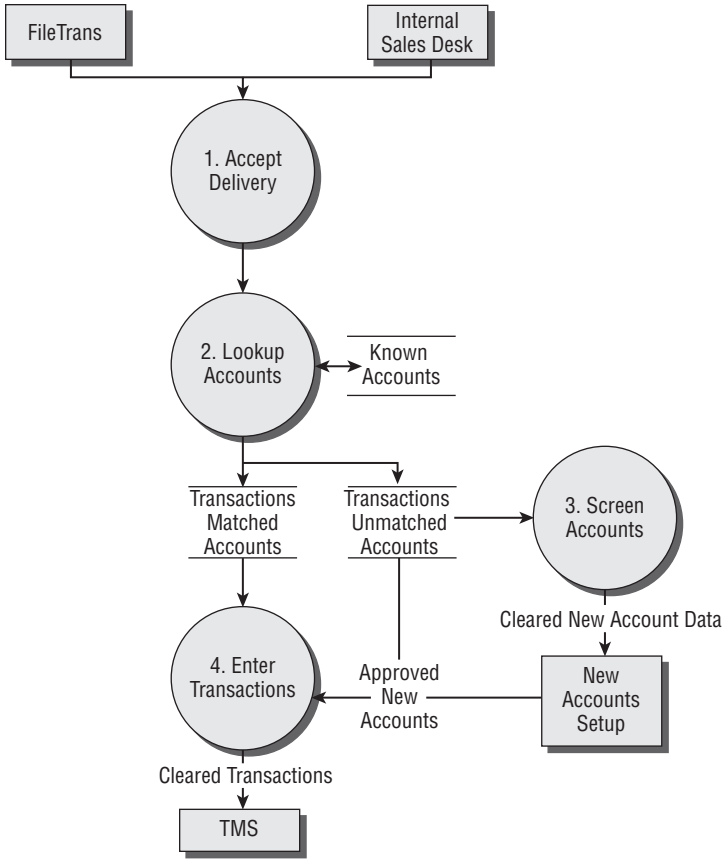
to be the point person to deal with IT. One of the senior managers thinks it would be a good idea to write what she calls a "business requirements document." Or perhaps IT has appointed you, the harried software development manager, to write what they call a "software requirements document."

Why Have We Turned from the Path of Righteousness?

So if writing a requirements document is such an obvious and important thing to do, and the consequences of not doing it are so severe, why is it often not done, or done very badly? This topic leads to an early digression. Knowing the answer to this question is not critical to learning how to write better requirements documents, but I have so often had to defend and justify the very necessity of the effort that I'm certain many of you will, too. It's helpful to understand how things got to be the way they are now and also to be ready to answer common protests about the process.

Let's begin with a brief history of writing software requirements. In the middle of the twentieth century, in the early days of the computer age, it was very expensive and time consuming to develop software. There weren't many engineers and there were few tools to automate their work. The wisest of the mid-century software makers, probably after some missteps, realized the need for writing clear and precise requirements before doing much development work. One of the best, Tom DeMarco, wrote a book named *Structured Analysis and System Specification*[2] that elegantly defined a standard for explaining software systems and writing requirements that came to be known simply as Structured Analysis. Structured Analysis explains systems as a series of processes that are represented in "Data Flow Diagrams." Mini-specs written in "Structured English" explain all of the process, and the "Data Dictionary" and "Data Structure Diagrams" describe and represent all of the system's data. There is more to it, of course, but these are the most important and basic elements. Structured Analysis was the primary way to do this work for most of the twentieth century, and in many ways it still is, especially in large organizations like corporations, government, and the military. Most of the ideas in this book are in some way part of Structured Analysis. The standard I recommend for data flow diagrams comes directly from Structured Analysis and is often called the *Yourdon DeMarco* data flow diagram. (Tom Demarco often partnered with Edward Yourdon, another giant in systems analysis.) An example follows.

[2] DeMarco, Tom. *Structured Analysis and System Specification*. Yourdon Press Series



When personal computers hit the mainstream, many more people became programmers and began writing software for all kinds of new purposes. New technologies, programming languages, and authoring tools made writing software much faster and easier. Analysis and documentation, however, were still slow and painful. New technologies and development methodologies challenged some of the basic tenets of Structured Analysis. If you could create working software in less time than it takes to write documentation, why bother? Just get something working as fast as possible and let your users actually use it instead of trying to decipher some long, tedious document. Small groups of engineers worked in garages and produced hugely successful software products. Developers sat next to their customers on trading floors and made changes in seconds. There was no time for requirements analysis on these scrappy teams. Developers often made software a bit at a time, releasing new versions with just a few new features every few months or days.

Now, as the pace of software development continues to speed up, the rigors of Structured Analysis are often considered too time-consuming. Various ways of developing software have evolved to keep up with how software is really being developed. These include Rapid Development, Agile, Extreme Programming, and others. To be fair, most credible developers recommend these less-structured methodologies only when developers can work very closely with their users and freely collaborate. Unfortunately, many projects throw out Structured Analysis (and all analysis, really) without meeting these conditions.

Most new methodologies include good-faith efforts to gather and document requirements. They describe various kinds of meetings or workshops that produce nonnarrative artifacts representing requirements: charts, diagrams, even Post-its. These artifacts are all perfectly clear to everyone in the room during the requirements meeting (except the guy who went to get snacks at a crucial moment). The downfall of these "documentation-lite" approaches is that they assume everyone can understand the artifacts produced by the minimal requirements-writing process and also that the consequences of building something that doesn't meet requirements can be easily rectified. If you weren't in the room when the consensus was reached (and perhaps even if you were) you may not have the exact same understanding of what is to be done as everyone else, and the artifacts may not communicate with enough precision to resolve the ambiguity. Three years later when everyone who worked on the last version has left the company (except the guy who went to get snacks during the requirements meeting), there is no intelligible record of what happened.

As software development becomes a global, commoditized process, requirements documents are again becoming more critical. In today's scattered work place, teams are often separated by oceans. As outsourcing becomes the norm, engineers are unhappily far away from their users and the close collaboration upon which many modern methodologies depend is often not possible. Increasingly, people who need software have to write requirements documents and send them to far-off engineering groups.

The speed of analysis has not kept up with the speed of development, and while programmers have been trained in legions around the world, we have not developed an equivalent population with the skills required to analyze and write requirements. Today we have an enormous need for this work, and few people who can do it. Often the job is imposed on development managers and project managers who lack training in analysis and communication. They

often produce far more than is necessary, agonizing for months, producing phone-book-length documents and wall-sized diagrams, and then wondering why their requirements aren't met.

Not surprisingly, many become disillusioned with the requirements process. I don't believe this is because writing requirements is an inherently flawed exercise. I think that the wrong people have been forced to do the work, they haven't been adequately trained or supported, and they usually waste a lot of effort working on the wrong content. People hate writing requirements when they do too much work and produce complex and confusing material.

A simple and flexible storytelling process can be satisfying and adaptable to a wide range of small and large efforts. This approach is even compatible with most of the latest methods of software development, including Agile, Extreme, and others.

I'll explain later who I think should be writing requirements. Now I'll get into how telling stories applies directly to the requirements process.

Can Stories Get Us Back on Track?

Perhaps while exploring many useful ways of representing software systems for engineers, Structured Analysis and other modern methods have become too abstract and moved away from how people ordinarily learn and communicate—too far away from storytelling. Storytelling is something everyone understands intuitively, immediately improving the process of gathering information and structuring the requirements document.

When you tell a story, you instinctively transform abstract knowledge into a logical structure. Whether writing or reading a story, you quickly get bored when you get into material that isn't interesting. You also can tell immediately when something sounds wrong, even if you haven't heard the story before. You get this very sophisticated validation process for free, without having to teach anything to anybody. Your mind is so well suited to stories that it almost automatically eliminates superfluous content and keeps you on the right subject. This quality of stories is especially useful when explaining very complex systems.

Let me tell another story to show how this works. When I switched from writing about software that regular people use to do everyday tasks, to writing about a credit risk system at an investment bank, I floundered for several days. The hardest part was that I couldn't see or use the system. All I could do was look at reports it produced and talk to the development manager, and he

wasn't much of a talker. The credit risk system was a series of jobs that ran at night, moving huge quantities of data around and then producing a lot data that credit analysts could query the next day. When the development manager would try to explain it to me (there was no real documentation), he would talk in great detail about what all the parts did and show me schema diagrams and other stuff I couldn't make sense of.

Fortunately, I was at the same time taking a class in analyzing information systems at night. I didn't know it, but I was learning Structured Analysis. The teacher never said anything about stories, but one night he said, "Every system is basically the same. It takes data from somewhere, it does something to it, and then it puts it somewhere else." We also learned a simple form of diagram. (See the example of a Yourdon DeMarco data flow diagram earlier in this chapter.) These diagrams showed data moving between what he called "processes." Each process was an action that changed data (not a thing). The lights went on.

The next day I asked my development manager to explain his system in a different way. "Let's say your system is a lumber mill. The data your system works with is like the wood the lumber mill processes. I want you to tell me the story of how your lumber mill makes wood products, starting with the trees in the forest and ending up with finish-grade plywood."

From then on, it was easy. We started with where all the data came from and I made him explain everything that happened to it until all the different kinds of finished data were saved at the end. Along the way he explained what was required at each step. I didn't write very many notes in prose as I had been doing previously, but drew diagrams with descriptive names for processes and data.

Thinking of what I was doing as writing a story made everything fall into place. Even if what I was producing looked pretty "techy," it moved along and made sense the same way a story does.

Making Structured Analysis into a Story

I've made some grandiose claims about the ability of telling stories to rescue the requirements-writing process. I've also discussed Structured Analysis and how it defines the basic tools of analysis that I believe are the best ways to approach requirements writing. Those of you who know something about Structured Analysis may puzzle over how I embrace these two approaches. The results of Structured Analysis are usually not considered a narrative. Many of

the strongest proponents of Structured Analysis (including Tom DeMarco himself, writing in *Structured Analysis and System Specification*) touted its advantages over what they called "narrative prose" analysis documents. I don't know exactly what they meant by that, but I can guess that they were criticizing the long, rambling documents I've seen that have a lot of text and no meaningful structure or systematic methodology. *Brain dump* is a more common and perhaps better term for documents like this.

Interestingly, near the end of *Structured Analysis*, DeMarco includes instructions for what he calls "packaging" the results of structured analysis. He doesn't call for any summary or narrative content to tie it all together. He does recommend including a guide to Structured Analysis that explains how all the tools work. He acknowledges that the methods of Structured Analysis will be new to many people. Instead of narrative about the content of the work itself, we get narrative about the tools. I don't think users reviewing requirements really want to know anything about Structured Analysis. This reminds me of those "How to use this book" sections I always find amusing. "Begin with Chapter 1 and turn each page to the left to proceed. Bold type means the content is important!" If a book *needs* a "How to use this book" section to be understood, it has already failed.

There is a middle ground, however, between rambling, unstructured "narrative" and nonnarrative collections of diagrams, specifications, and data dictionaries that only those who know Structured Analysis can understand. My "story" approach takes simplified versions of the most basic Structured Analysis tools and puts them in a narrative structure that explains itself as it goes along. There's no need for a separate "How to read a story" section.

One key advantage of Structured Analysis, according to its authors, was that parts could stand on their own, be distributed for review independently, created out of order, and didn't have to be read as a monolithic phone book.

The parts of a good narrative document can also be created out of order and read on their own. In fact, I recommend starting with data flow diagrams. A bit of narrative content that explains how a diagram fits into the overall system makes it *more* capable of being understood on its own, not less. It's very easy for readers to skim or skip around in a well-structured document in which every section explains how it fits into the whole. One of the biggest problems I've observed when analysts distribute parts of their "nonnarrative" work is they leave off meaningful titles and there is not a brief summary that explains the basic purpose of the current part.

I can't tell you the number of times I've had to strenuously search through carefully crafted, richly detailed diagrams and code descriptions trying to figure out the basic purpose and name of the system under discussion.

My favorite example was a 400-page functional spec for an equity trading system that did not say anywhere in the entire document that the system was for trading equities. I was trying to prove to regulators that the system was documented, and for lack of about six words ("TraderX is for trading equities") the 400 pages were useless as evidence.

Narrative does not have to be rambling, incoherent, and unstructured. It can be as clear and precise as any diagram, table, equation, or code. It just has to be done right. Diagrams, tables, equations, and code can be a lot clearer and more compelling with just a little bit of good old-fashioned narrative support. Disparagers of the power of narrative are usually just bad at it, or they've never been taught.

To win over the nonnarrators, I have to broaden their ideas about narrative. Narrative doesn't have to be long paragraphs, few headings, and no pictures. You can tell a story in pictures, text, video, audio, dance, or any other media, and you can mix and match as you see fit. The key thing is that the media are effective in getting your point across, and that there is logical, meaningful structure to the material in which the parts build on each other from beginning to end.

In creating an effective narrative form for a software requirements document, I use a mix of very specific elements:

- Clear, precise words
- Clear, short sentences
- Clear, short paragraphs
- A strong overall structure that breaks up all the content into short sections with meaningful titles, organized by subject
- Lots of diagrams (data flow diagrams, to be specific) showing every process in the system
- Structured descriptions of each process
- Descriptions of all the important data in the system
- Summary material that ties it all together

The result doesn't look much like what you think of as a story, but if you pick it up and turn to the first page, you'll find an actual beginning that states the problem, proposes a solution, and then compels you to keep reading. You

can go page by page, or skip to the part you care about. Wherever you go, you'll be able to clearly observe how the story has continued. You'll know where you are, what subject is being addressed, and how it fits into the whole. You'll also know where to go to find anything else.

The tools comprising Structured Analysis do not produce what people think of as a story, but there is a story underlying nearly every part—it just doesn't look like an old-fashioned one. Add a bit of summary material before your diagrams and put them in order, and you have a graphic story, more like a comic book than a traditional book, but a story nonetheless.

Who Should Do This Work?

Most of the people I've heard strongly criticize writing software requirements are really good at developing software or managing projects and really bad at writing. With better training and support, some people who feel this way can produce adequate requirements, but there is no escaping their fundamental distaste for the work. After much strenuous effort to work around and with those who hate writing requirements, I've come to agree with them. Great software developers and project managers have many wonderful talents, but writing usually isn't one of them.

Why not get people who are really good writers to do it? When you know Beethoven and Shakespeare, you don't force Beethoven to write you a sonnet; you call Shakespeare. Why have we been forcing software developers to do a writer's job? Where are all the requirements writers?

There aren't many working in IT departments, consulting firms, and software companies. These organizations have become so focused on what they consider "technical" skills that you will have a hard time finding anyone who can write in many of them. I became frustrated with the recruiting process at a firm where I used to work when I found that its college recruiting targeted only computer science majors. I asked one of the recruiters what percentage of the IT jobs at the firm were best filled by computer science majors. The answer was maybe 40 percent, definitely less than half. What about all the other jobs? No answer. I have nothing against computer science majors, but when I interviewed them, only a few had any English or even liberal arts classes on their transcripts.

Why isn't writing English considered an important technical skill? Who is going to write the documentation, the proposals, the presentations, the

announcements, and the e-mails, let alone the software requirements? There are now far more people who can write good Java code working in technology than can write a clear sentence. When it comes time to write requirements, it honestly doesn't occur to many people working in technology to go find a writer; they've never met one, and there aren't any on staff.

Education has been emphasizing science, math, and engineering for over a generation and writing has been devalued as a professional skill, left to liberal arts majors, poets, novelists, journalists, and other miscreants.

There are, however, a few hardy souls hidden in technology companies and IT departments who can skim this book and report for duty tomorrow, ready to write requirements better than most of the people doing the work now. Strangely, many of these people do not know what software requirements are. They are called *technical writers*. Most often they write end-user documentation: online help systems, manuals, website content, runbooks, and the like. The best are those writing technical documentation for engineers: functional specifications, programming manuals, operations guides, and so on.

Good technical writers usually begin with strong writing skills and have learned enough technology to work well with engineers and developers to communicate effectively with a wide range of end users and technology people. Many have participated in the software development process enough to understand much of the nuance and language. They have worked on many projects and are good at interviewing subject matter experts and learning technology quickly. They can also write a lot faster and better, and therefore cheaper, than software developers or business users. Of course they do not start out with the information in their heads, so it may seem like more time is required for interviews and subsequent reviews. But overall, technical writers require much less of important people's time than developers do in writing requirements. (As a former technical writer, I have to confess a strong bias in discussing our advantages in the requirements process.)

Aside from software development managers and project managers, the most common title for writers of requirements is *business analyst* or *requirements analyst*. These analysts come from a wide variety of backgrounds. Some are former developers who found they liked the work better than programming, some were project managers, and some worked in a line of business and managed the relationship with the software developers. Some of these analysts are excellent at their jobs, but most are better at what they started out doing than they are at writing and communicating.

With better support and training, many of the people pressed into service writing requirements today, regardless of education or background, could do a better job, and many people who aren't doing it at all could be taught.

If you're one of those having this work imposed on you and you really think you're the wrong person for the job, you should have no shame in making it known. Tell your boss the firm could save money and free you up to do what you're good at if it would hire a technical writer to write requirements.

If you can't get a technical writer or a good business analyst, this book should be helpful to you as a starting point. If you don't like to write or think you are bad at writing, I encourage you to write as little as possible. The most text-heavy documents I've seen were written by engineers and included many pages of superfluous material. Follow the guidelines in this book and stick to short, punchy sections, paragraphs, and sentences. Use lots of diagrams and if you see more than an inch or two of text anywhere, break it up. Remember, a comic book is usually a very good story.

Summary

My advice breaks down to just a few essential points. First there are some very general ones, some of which we've gone into already:

- Stories are the best way to communicate requirements.
- We approach writing requirements like you are telling a story.
- Story elements such as conflict, theme, plot, and character are directly analogous to the requirements process.
- Software requirements explain a business problem and the requirements of a software solution in nontechnical language.
- Software requirements do not include project management details such as schedule, resources, and cost.
- There is no well-understood distinction between *software* requirements and *business* requirements.
- Structured Analysis defined the basic tools for specifying requirements in 1979 and has remained an important methodology ever since.
- Structured Analysis did not include a narrative structure to hold all of its parts together.
- Combining the basic elements of Structured Analysis with a simple narrative structure results in a much more effective final document.
- Technical writers are best suited to writing requirements, but others can adequately do the work with some training and practice.

Having discussed these general ideas about requirements and stories, I'll thoroughly describe specific skills and tools, beginning with three skills that may be new to you:

- Write clear and precise words, sentences, and paragraphs that specify testable requirements.
- Create data flow diagrams simple enough for anyone to follow, but sufficiently rigorous and precise to be useful to engineers.
- Explain processes systematically to capture all the vital information and requirements.
- Associate requirements with processes. Write requirements using precise terms and with a measurable outcome.
- Create a system inventory that includes every element of the system: every known process, actor, output, report, category of data, and anything that the team thinks is important about the system.
- Break down the system into a series of events or processes so that they form a linear story of what happens at every step.
- Structure this information in a logical sequence, including an executive summary, the current state, future state, data descriptions, and various summary sections and optional appendixes for handling specialized material specific to certain projects.
- Maintain and reuse requirements content when writing test plans, user documentation, functional specifications, and other materials.

The rest of this book covers this advice in considerably more detail. The next chapter starts where the story begins, with clear and precise words, sentences, and paragraphs.